Universidade Federal de Roraima
Departamento de Ciência da Computação
Arquitetura e Organização de Computadores

PROJETO FINAL
PRAZO DE ENTREGA: 13/01/2026

ALUNO(A):_____

ATENÇÃO: Vale ressaltar as instruções apresentadas aqui já foram apresentadas em aula, logo o foco deste documento é na descrição dos artefatos a serem entregues para o projeto final. Assim, descreva:

1. As soluções com o máximo de detalhes possível inclusive a forma como os testes foram feitos;
2. Todos os artefatos: relatório (no mínimo 4 páginas), código fonte de programas, e outros gerados para este trabalho devem ser adicionados em um repositório no site github.com, com o seguinte formato **`AOC_Nome1Nome2_UFRR_2025;`**
3. O modelo de relatório do projeto deverá seguir o formato da SBC de artigos, disponível em https://www.sbc.org.br/wp-content/uploads/2024/07/modelosparapublicaodeartigos.zip **;**
4. O referido repositório deve ser adicionado na atividade (incluindo a URL com extensão .git) do projeto final no SIGAA no tópico de aula **Apresentação do Projeto Final - Parte 1 (13/01/2026 - 13/01/2026)**;
5. O projeto deve apresentar a IDE utilizada e como o compilar/executar o processador do projeto final.

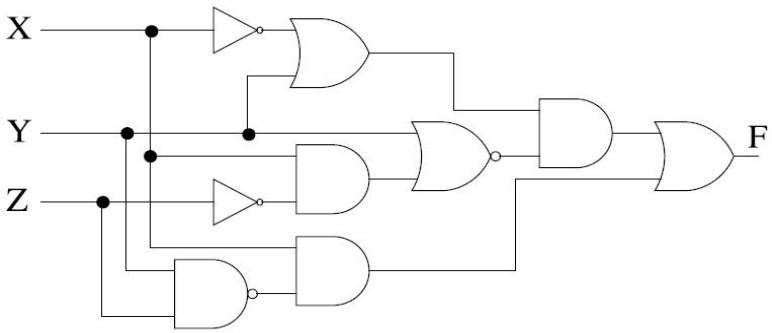
[TASK 01] Projetar e implementar um processador RISC de **8 bits multiciclo** (semelhante ao MIPS), segue os requisitos para a elaboração deste projeto:

1. Os componentes do processador deverão ser escritos na linguagem de programação VHDL;
2. A descrição da estrutura das instruções suportadas pelo processador deverá ser apresentada por classe e suas respectivas divisões por bits;
3. A descrição da linguagem (assembly e binário) suportadas pelo processador, como no caso do MIPS;
4. Apresentação do *datapath* (barramento com suas conexões) do processador indicando a quantidade de bits por trilhas e as entradas e saídas para cada componente;
5. Apresentação da unidade de controle e os sinais de controle para cada instrução do processador;
6. As seguintes instruções são obrigatórias para o processador: load, store, soma, subtração, beq, salto incondicional; e
7. Apresentar simulações e testes usando *waveforms ou testbench* para cada instrução e pelo menos um programa utilizando todas as instruções suportadas pelo processador.

[TASK 02] Análise e Verificação de Circuitos Lógicos com Fórmulas Booleanas e Z3

Nesta atividade, deve-se:

1. **Analisar os seguintes circuitos digitais e formulas lógicas:**

(A) 
(B) Somador completo de 8 bits
(C) Unidade de controle (codificada em uma Máquina de Estados Finitos Completa) do processador multiciclo MIPS
(D) F = ABCD+ABC'D+ABC'D'+AB'CD+A'BCD+A'BCD'+A'BC'D+A'B'C'D
(E) Unidade Lógica e Aritmética (ULA) de 8 bits com as funções de substração, XOR, NAND, NOR e shift de 2 bits a esquerda
(F) Fluxo de execução de uma instrução do tipo R no MIPS de 32bits.

2. **Converter os circuitos para fórmulas booleanas:** Representar matematicamente o funcionamento do circuito. Para cada circuito, os deve-se:
    1. Identificar a função lógica de cada porta.
    2. Escrever a expressão booleana final que representa o circuito. Exemplo: Para um circuito com $F=(A \land B) \lor (\neg C)$, a expressão booleana será escrita.

3. **Verificar as especificações:** Utilizar o *solver* Z3 (instalável via `pip install z3-solver`) em Python 3.x para verificar se o circuito atende a condições específicas, como equivalência lógica, validade de saída, ou redundância de elementos. Assim, deve-se definir variáveis booleanas para as entradas e utilizar as funções do Z3 para modelar o circuito lógico.

- **Validação de saída:** Certificar-se de que a saída atende a condições específicas. Exemplo: $F=(A \land B) \lor (\neg C)$

```
from z3 import *

# Definir variáveis
A, B, C = Bools('A B C')

# Expressão booleana do circuito
F = Or(And(A, B), Not(C))

# Solver para verificar a especificação
solver = Solver()
```

```
# Especificação: A saída deve ser verdadeira se A = True, B = True, C = False
solver.add(Not(F))  # Verificar se F é falso para a condição especificada
solver.add(A == True, B == True, C == False)

# Resultado da verificação
if solver.check() == sat:
    print("A especificação falhou.")
else:
    print("A especificação foi atendida.")
```

- **Redundância de componentes:** Verificar se partes do circuito podem ser removidas sem alterar a saída. Exemplo:

  Fórmula Booleana do Circuito Completo: $F=((A \wedge B) \vee C) \vee C$

  Note que C está presente duas vezes na fórmula, sugerindo que a segunda ocorrência pode ser redundante.

- **Forma Simplificada (Manual):**

  Aplicando simplificação lógica:

  1. Pela **propriedade de idempotência** da porta OR ($X \vee X=X$): $F=(A \wedge B) \vee C$

  Ou seja, a segunda porta OR é redundante.

  Passos da verificação:

  1. Modelar o circuito completo no Z3.
  2. Modelar o circuito simplificado no Z3.
  3. Verificar se ambas as saídas são **equivalentes** para qualquer conjunto de entradas.
  4. Caso sejam equivalentes, a redundância é confirmada.

```
from z3 import *

# Definir variáveis booleanas
A, B, C = Bools('A B C')

# Circuito original (com redundância)
F_original = Or(Or(And(A, B), C), C)

# Circuito simplificado (sem redundância)
F_simplificado = Or(And(A, B), C)

# Solver para verificar equivalência
solver = Solver()

# Adicionar condição de inequivalência (se forem diferentes, há um problema)
solver.add(F_original != F_simplificado)

# Verificar
if solver.check() == sat:
    print("Os circuitos NÃO são equivalentes. A redundância não pode ser
removida.")
else:
    print("Os circuitos são equivalentes. A redundância pode ser removida.")
```

Explicação do Código

1. **Modelagem do circuito original:** O circuito completo é representado na fórmula $F_{original}=((A \wedge B) \vee C) \vee C$.
2. **Modelagem do circuito simplificado:** A fórmula simplificada é $F_{simplificado}=(A \wedge B) \vee C$.
3. **Condição de inequivalência:** Adicionamos $F_{original}/F_{simplificado}$ para que o Z3 tente encontrar uma entrada que cause diferença nas saídas.
4. **Interpretação do resultado:**
    - Se **SAT (satisfatível):** Há entradas que diferenciam os circuitos, e a redundância não pode ser removida.
    - Se **UNSAT (insatisfatível):** Os circuitos são equivalentes para todas as entradas, confirmando que a redundância pode ser eliminada.

» Entrega Final

Deverá ser entregue:

1. Fórmulas booleanas derivadas dos circuitos.
2. Código Python utilizando o Z3.
3. Relatório descrevendo os resultados das verificações (Validação de saída, Redundância de componentes e Forma Simplificada) realizadas.

[TASK 03] Otimização de Códigos MIPS em Processadores com Pipeline

Objetivo

- Compreender como a organização pipeline de um processador influencia a execução de instruções em MIPS.
- Utilizar Python para identificar e otimizar gargalos no código MIPS, como dependências de dados e bolhas no pipeline.

Descrição:

1. **Analisar o problema**: Estudar um trechos de códigos MIPS fornecido, identificar dependências e ineficiências que afetam o desempenho no pipeline.
2. **Automatizar com Python**: Desenvolver um script Python que:
    - Detecte dependências de dados.
    - Insira instruções NOP (se necessário).
    - Execute o desdobramento de loops
    - Reorganize instruções para minimizar bolhas e melhorar o *throughput*.
3. **Comparar desempenho**: Avaliar o número de ciclos antes e depois da otimização.

Etapas da Atividade:

1. **Estudo Teórico**
    - Revisar o funcionamento do pipeline e os tipos de dependências (dados: RAW, WAR, WAW e controle: execução de *loops*).
    - Discutir como reorganizar instruções pode reduzir bolhas no pipeline.

2. **Códigos MIPS a serem analisados:**

(A)
`ADD $t0, $t1, $t2`
`SUB $t3, $t0, $t4`
`AND $t5, $t3, $t6`
`OR  $t7, $t5, $t8`
	(B)
	`ADD R1,  R2, R3`
	`SUB R4,  R1, R5`
	`AND R6,  R1, R7`
	`OR  R8,  R1, R9`
	`XOR R10, R1, R11`

(C)
`Loop:`
`  L.D  F0, 0(R1)`
`  ADD.D  F4, F0, F2`
`  S.D  F4, 0(R1)`
`  DADDUI R1, R1, #-8`
`  BNE  R1, R2, Loop`
	(D)
	`DIV.D F0, F2, F4`
	`ADD.D F10, F0, F8`
	`SUB.D F12, F8, F14`

(E)
`DIV.D F0, F2, F4`
`ADD.D F6, F0, F8`
`S.D F6, 0(R1)`
`SUB.D F8, F10, F14`
`MUL.D F6, F10, F8`
	(F)
	`ADD R4, R5, R6`
	`BEQ R1, R2, EXIT`
	`OR  R7, R8, R9`

3. **Script Python para otimizações de códigos MIPS:**

- Leia o código MIPS a partir de um arquivo texto.
- Identifique dependências de dados entre as instruções.
- Proponha inserções de NOP, reordenações de instruções, o desdobramento de loops para minimizar bolhas. Exemplo de estrutura para o script Python (observação o SCRIPT deve ser alterado de acordo com as otimizações solicitadas):

```
def detect_dependencies(instructions):
    """Detecta dependências de dados entre instruções."""
    dependencies = []
    registers = {}

    for idx, instr in enumerate(instructions):
        parts = instr.split()
        dest = parts[1].strip(",")
        src = parts[2:]

        # Verifica dependências
        for reg in src:
            if reg in registers and registers[reg] > idx:
                dependencies.append((idx, registers[reg]))

        registers[dest] = idx

    return dependencies

def optimize_pipeline(instructions):
    """Reorganiza ou insere NOPs para minimizar bolhas."""
    optimized = []
    for instr in instructions:
        optimized.append(instr)
        # Simulação: insira NOP para cada dependência (simplificação)
        optimized.append("nop # Resolvendo dependência")

    return optimized

# Código MIPS inicial
mips_code = [
    "add $t0, $t1, $t2",
    "sub $t3, $t0, $t4",
    "and $t5, $t3, $t6",
    "or  $t7, $t5, $t8"
]

print("Dependências detectadas:", detect_dependencies(mips_code))
print("\nCódigo otimizado:")
for instr in optimize_pipeline(mips_code):
    print(instr)
```

4. **Teste e Avaliação**

- Executar o script e observar como o código foi reorganizado.
- Discutir como a otimização afeta o desempenho em um pipeline típico de 5 estágios (IF, ID, EX, MEM, WB).

Critérios de Avaliação

- Identificação correta de dependências de dados.
- Funcionamento do script Python.
- Qualidade da reorganização e redução de bolhas.
- Documentação e clareza no código Python.

[TASK 04] Automatizando e ampliando a metodologia de verificação (VHDL → C → ESBMC)

O artigo **Open Technologies in Formal Verification: Transforming Code for Circuit Validation** propõe duas estratégias de tradução (direta VHDL→C e múltipla VHDL→Verilog→C) e mostra que a abordagem em dois passos (VHDL→Verilog→C) foi mais flexível, mas há limitações de cobertura de constructs VHDL e necessidade de ferramentas auxiliares e instrumentação robusta para assertions e nondeterminismo. Esta atividade visa investigar ferramentas auxiliares que permitam automatizar e ampliar cobertura e habilitar testes baseados em propriedades de forma totalmente automatizada.

Objetivos da atividade:

1. Identificar limites do pipeline com exemplos que usem downto, integer, arrays e processos sensíveis a clock.
2. Estender o pipeline com um passo auxiliar: usar ferramantes como Yosys (para Verilog elaboration / assertions synthesis) ou scripts de pré-processamento para aumentar compatibilidade.
3. Integrar o SymbiYosys como passo adicional de verificação para complementar ESBMC (trabalhar dualmente com BMC e SAT/SMT).
4. Entregar um script de automação (Python) que: detecta arquivos VHDL, gera arquivo auxiliar (inputs/outputs/pre-post), executa tradutores, injeta assertions (tags @c2vhdl:ASSERT), instrumenta C gerado, chama ESBMC e coleta resultados.
5. Criar um Front-end unificador com AST comum — construir um frontend que parseie VHDL em AST (ex.: usando ghdl + export AST) e gere Verilog/C com regras consistentes, reduzindo erros introduzidos por múltiplas ferramentas.